Angular

Testing

Expliqué par

David Gilson

https://github.com/gilsdav

Assets

- https://github.com/gilsdav/af-testing-workshop
- Angular CLI 8

Sommaire

- Types de tests
- Angular et le testing
- Structure de test
- Comment tester
  - Service Business
  - Pipe
  - Component
  - Service Web
  - Routing
  - Store

Types de test

- **Unitaire** : partie de l'application complètement isolée
- **Intégration** : regroupement de plusieurs parties
- **E2E** : flux complet de l'application

Angular et le testing

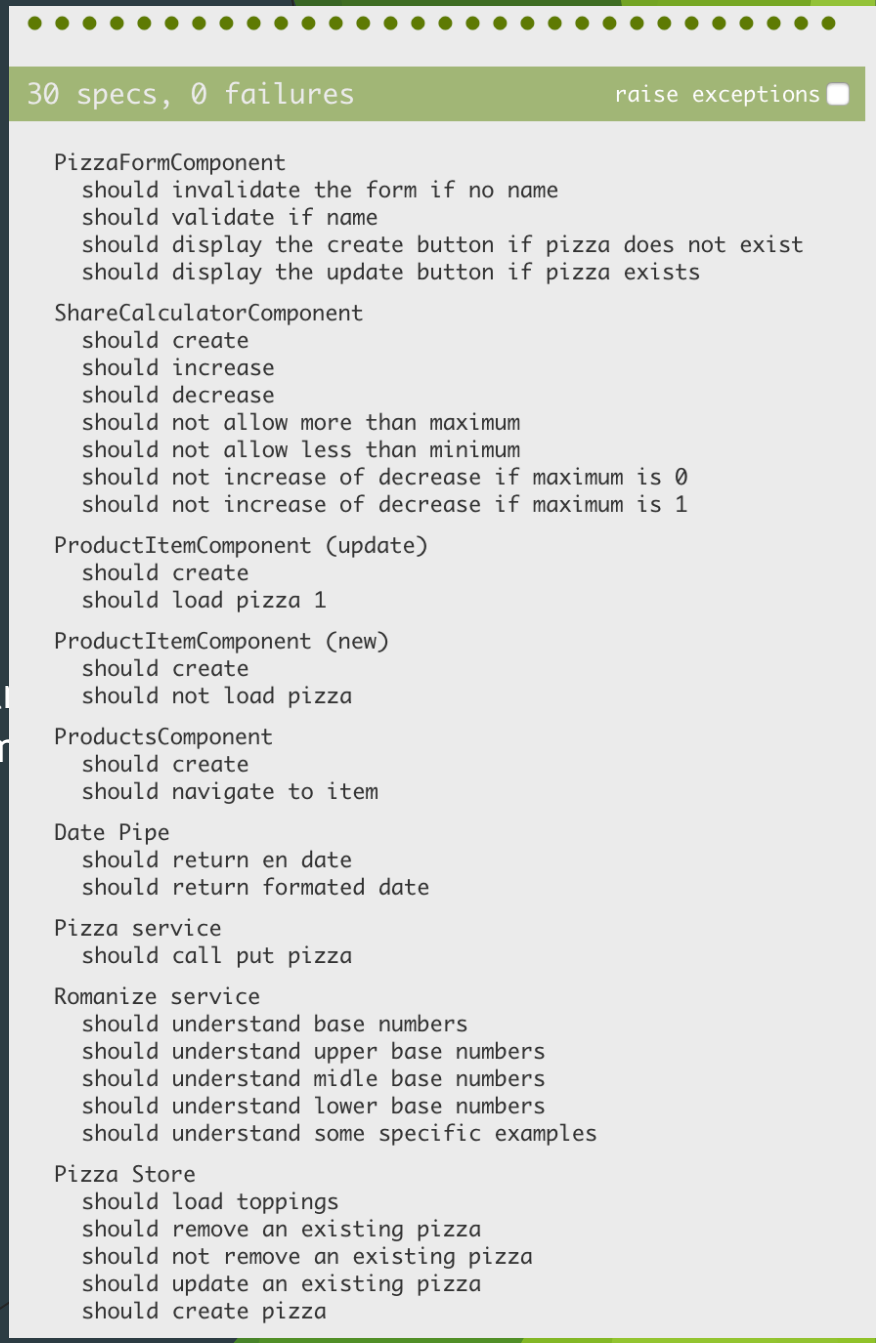
- Angular a été implémenté pour être complètement testable
- Il possède des APIs dédiés au test: **HttpTestingModule**, **RouterTestingModule**, **TestBed**, etc...
- Le CLI configure automatiquement l'écosystème de test

Outils utilisés

- **Karma**: Exécution des tests
- **Jasmine**: Framework JavaScript pour écrire des tests en BDD
  - Behaviour Driven Development veut que la description des tests soit dans un format lisible par l'homme pour permettre au non développeurs de comprendre ce qui est testé.
- **Protractor**: Framework de test E2E

Outils utilisés

- **Karma**: Exécution des tests
- **Jasmine**: Framework JavaScript pour écrire des tests en BDD
  - Behaviour Driven Development veut que la description des tests soit dan format lisible par l'homme pour permettre au non développeurs de com qui est testé.
- **Protractor**: Framework de test E2E



Lancer les tests

Utilisez les scripts existants dans le package.json

- npm run **test**
- npm run **e2e**

Structure de test

- **Suite de tests**: ensemble de spécifications
  - **describe()**

```
describe('Testsuite', () => {

  // spécifications

});
```

- **Spécification**: spécification d'un scénario de test
  - **it()** + **expect()**

```
describe('Testsuite', () => {

  it('should work', () => {
    expect(true).toBe(true);
  });

});
```

Vérifications

- La fonction expect() doit être chainée avec des « **matchers** »

```
expect(actual).toBe(expectation) // ===
expect(actual).toBeDefined() // not undefined
expect(actual).toBeFalsy() // falsy
expect(actual).toBeGreaterThan(value) // >
expect(actual).toEqual(expectation) // deep equality
...
```

Préparation d’un suite de tests

- Il est possible de préparer une suite de tests à l’aide des fonctions

```
beforeEach(() => {
});

afterEach(() => {
});
```

Tester un service business

- Si le service ne nécessite pas ou peu de dépendances, le plus simple et efficace est de créer une instance nous même.

```
import { MetaService } from './meta.service’;

describe('Service: Meta', () => {
  let service: MetaService;

  beforeEach(() => {
    service = new MetaService();
  });

  ...
});
```

Tester un service business

- Il est tout aussi simple de tester le retour d'une fonction

```
it('should return http://test.net', () => {
  expect(service.getHost('http://test.net/bonjour')).toBe('http://test.net');
});
```

- Nous pouvez également informer Karma qu'il y a une action asynchrone et quand elle se termine à l'aide du paramètre « done »

```
it('should return http://test.net from observable', (done: DoneFn) => {
  service.getHostAsync('http://test.net/bonjour').subscribe(value => {
    expect(value).toBe('http://test.net');
    done();
  });
});
```

Exercice

Tester un service business
Dépendances

- Créer une nouvelle instance

```
beforeEach(() => {
  service = new MetaService(new MyDependency());
});
```

- Faire un stub du service

```
class MyFakeDependency {
  public sameMethodsInDependency() {
    ...
  }
}

beforeEach(() => {
  service = new MetaService(new MyFakeDependency() as MyDependency);
});
```

Pipe

- Un pipe se test de la même façon que le service business avec la différence qu'il n'y a que la fonction « transform » à tester.

Exercice

Tester un composant

- Tester un composant visuel est simple mais le fait de devoir accéder à la partie HTML du composant demande plus d'outils:
  - **TestBed:** Configurer et initialiser un environement destiné aux tests
  - **fakeAsync:** Permet de simuler un passage dans le temps (exemple: simuler le fait que nous avons attendu 5 secondes)
  - **Testing Modules:** HttpClientTestingModule, RouterTestingModule, ...etc.

Tester un composant
TestBed

```
import { TestBed, ComponentFixture } from '@angular/core/testing';

describe('ContactsDetailComponent', () => {

  const fixture: ComponentFixture<ContactsDetailComponent>;
  const component: ContactsDetailComponent;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ContactsDetailComponent],
      imports: [ContactsMaterialModule]
    });

    fixture = TestBed.createComponent(ContactsDetailComponent);
    component = fixture.componentInstance;
  });
  ...
});
```

Tester un composant
Queries et Detect Change

- **fixture.detectChanges():** Actionner un change detection
  - Il appellera ngOnInit au premier appel dans un « it » (ou beforeEach)

- **fixture.debugElement.query():** Requêtes dans le DOM
  - fixture.debugElement.query(By.css('p'))

Tester un composant
Queries et Detect Change

```
import { By } from '@angular/platform-browser';

it('should render contact', () => {

  const expectedContact = {...testContact};
  component.contact = expectedContact;

  const debugEl = fixture.debugElement.query(
    By.css('mat-card-title')
  );

  fixture.detectChanges();

  expect(debugEl.nativeElement.textContent)
    .toContain(expectedContact.name);
});
```

Tester un composant
Input/Output

- **Input**
  - Utiliser la variable comme si ça n'était pas un input
    - **component.pizza = { id: 1 } as Pizza;**
  - Exécuter la fonction « ngOnChange » si besoin
    - **component.ngOnChanges({ pizza: component.pizza } as any);**
- Output
  - Souscrire à l'output

```
it('should emit back event', () => {

  let backEmitted = false;
  const buttonEl = fixture.debugElement.query(...);

  component.back.subscribe(() => {
    backEmitted = true;
  });

  buttonEl.triggerEventHandler('click', null);
  expect(backEmitted).toBe(true);
});
```

Exercice

Tester un composant avec formulaire

- Initialisez les valeurs de votre formulaire
  - component.form.**patchValue**({name: ‘Dav’});

Exercice

Web services

- HttpClient nous offre une API de testing: **HttpClientTestingController**

```
let backend: HttpTestingController;
let service: ContactService;

beforeEach(() => {
  TestBed.configureTestingModule({
    imports: [HttpClientTestingModule],
    providers: [ContactService]
  });
  service = TestBed.get(ContactService);
  backend = TestBed.get(HttpTestingController);
});
```

Web services

- Lancer l'appel et vérifier que la réponse est bien celle attendue

```
service.getAll().subscribe(response => expect(response).toEqual(mappingResult));
```

- Ecouter les appels sur l'URL "/users" (celle exécutée à la ligne précédente)

```
const req = backend.expectOne('/users');
```

- Vérifier que la méthode est bien celle attendue

```
expect(req.request.method).toEqual('GET');
```

- Envoyer des données mockées au service

```
req.flush(serviceResponse);
```

- Vérifier que l'appel services lancé a bien été vérifié

```
backend.verify();
```

Web services

- Il est possible de tester plusieurs appels avec des réponses différentes en remplaçant« expectOne » par la méthode « match » qui va alors renvoyer une liste de requêtes à la place d'une seule

```
const requests = backend.match('/users');
expect(requests.length).toEqual(2);
requests[0].flush({ users: [] });
requests[1].flush({ users: [{ name: 'hello' }] });
```

- Pour tester la réaction aux erreurs, il est également possible d'en mocker en spécifiant un code d'erreur ou non

```
// Simulation d'une erreur ayant un code d'erreur
req.flush('message', {status: 404, statusText: 'Not Found'});
// Simulation d'une erreur n'ayant pas de code d'erreur
req.error(new ErrorEvent('Network error', { message: 'message' }));
```

Exercice

Routing
Espions

- Si l'on veut tester qu'une navigation a été lancée suite à une action, nous pouvons utiliser les **espions**
  - Un espion permet de vérifier si (ou combien de fois) une fonction a été appelée
  - Par défaut il annule le comportement de la fonction
  - Peut servir pour mocker le comportement de la fonction

```
it('should use spy', () => {
  const foo = {
    setBar: function (value) {}
  };

  spyOn(foo, 'setBar');

  foo.setBar(123);
  expect(foo.setBar).toHaveBeenCalled();
  expect(foo.setBar).toHaveBeenCalledWith(123);
});
```

Routing
Espions

```
spyOn(contactsService, 'getContacts').and.returnValue(of([
  { id: 0, name: 'First contact', image: '/assets/images/1.jpg' },
  { id: 1, name: 'Second contact', image: '/assets/images/2.jpg' }
]));
```

```
spyOn(contactsService, 'getContacts').and.callThrough();
```

Routing
Espions

```
it('redirect to other page', () => {
  const link = fixture.debugElement.query(By.css('#navigate')).nativeElement as HTMLLinkElement;

  // Ajouter un espion sur la méthode "navigateByUrl" du router
  const spy = spyOn(router, 'navigateByUrl');

  link.click();

  // Vérification que la méthode a été appelée 1 fois
  expect(spy.calls.count()).toBe(1);

  // Récupération du premier paramètre fourni
  const url = spy.calls.first().args[0].toString();

  // Vérification que l'URL appelée est bien "/other"
  expect(url).toContain('/other');
});
```

Routing

- **RouterTestingModule:** API de test de routes qui offre des espions sur les APIs comme Location ou LocationStrategy

```
const APP_ROUTES: Routes = [
  { path: '', component: SomeComponent },
  { path: 'bar', component: BarComponent }
];

TestBed.configureTestingModule({
  declarations: [
    ContactsDetailViewComponent,
    ContactsListComponent,
    ContactsEditorComponent
  ],
  imports: [
    HttpClientModule,
    RouterTestingModule.withRoutes(APP_ROUTES)
  ]
  schemas: [NO_ERRORS_SCHEMA]
});
```

Routing

- Stub de route

```
const activatedRouteStub = {
  snapshot: {
    params: { id: '2' }
  },
  params: new BehaviorSubject({id: 2})
};
```

```
{ provide: ActivatedRoute, useValue: activatedRouteStub }
```

```
it('should fetch contact by given route param', () => {
  ...
  fixture.detectChanges();
  expect(contactsService.getContact).toHaveBeenCalled();
  expect(contactsService.getContact).toHaveBeenCalledWith('2');
  expect(component.contact).toEqual(expectedContact);
});
```

Exercice

Store

- NGXS fournis une fonction « reset(…) » qui permet d'initialiser l'état du store à celles nécessaires pour le test.

```
export const SOME_DESIRED_STATE = {
  animals: ['Panda'],
};

describe('Zoo state', () => {
  let store: Store;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      imports: [NgxsModule.forRoot([ZooState])],
    }).compileComponents();

    store = TestBed.get(Store);
    store.reset(SOME_DESIRED_STATE);
  }));

  it('should toggles feed', () => {
    store.dispatch(new FeedAnimals());
    store.selectOnce(state => state.zoo.feed).subscribe(feed => {
      expect(feed).toBe(true);
    });
  });
});
```

Exercice

FakeAsync

- Simuler un passage de temps pour « setTimeout », « setInterval » ou « requestAnimationFrame »
  - tick(x) : simuler un passage de x millisecondes
  - flush() : simuler un passage de temps jusqu'à ce que la queue des taches macro soient vide

```
it('fakeAsync works', fakeAsync(() => {
  const promise = new Promise(resolve => {
    setTimeout(resolve, 10);
  });

  let done = false;
  promise.then(() => (done = true));

  tick(50);
  expect(done).toBeTruthy();
}));
```

Marble (observables)

```
import { cold, getTestScheduler } from 'jasmine-marbles';
```

```
const q$ = cold('---x|', { x: mockedData });
```

- Attends 3 frames (---) puis émet la valeur (x) et enfin se complète (|)
- Utilisez un « # » pour générer une erreur (---#|)
- Plusieurs valeurs (---x---y|)

```
getTestScheduler().flush();
```

- Flush de l'observable (démarrage du processus)

Exercice

Références

- https://blog.thoughtram.io
- https://wetry.eu
- https://alligator.io/angular/testing-async-fakeasync/
- https://angular.io/guide/testing